UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

Práctica 3: Abstracción. TDA Vídeo

(Práctica puntuable)

Dpto. Ciencias de la Computación e Inteligencia Artificial
E.T.S. de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Estructuras de Datos

Grado en Ingeniería Informática
Doble Grado en Ingeniería Informática y Matemáticas
Doble Grado en Ingeniería Informática y ADE

1.- Introducción

1.1 Imágenes de niveles de gris

Una **imagen digital** de niveles de gris puede verse como una matriz bidimensional de puntos (píxeles, en este contexto) en la que cada uno tiene asociado un nivel de luminosidad cuyos valores están en el conjunto {0,1, ..., 255} de forma que el 0 indica mínima luminosidad (negro) y el 255 la máxima luminosidad (blanco). Los restantes valores indican niveles intermedios de luminosidad (grises), siendo más oscuros cuanto menor sea su valor. Con esta representación, cada píxel requiere únicamente un byte (**unsigned char**). En la figura siguiente mostramos un esquema que ilustra el concepto de una imagen con **alto** filas y **ancho** columnas. Cada una de las alto x ancho celdas representa un píxel o punto de la imagen y puede guardar un valor del conjunto {0,1, ..., 255}.

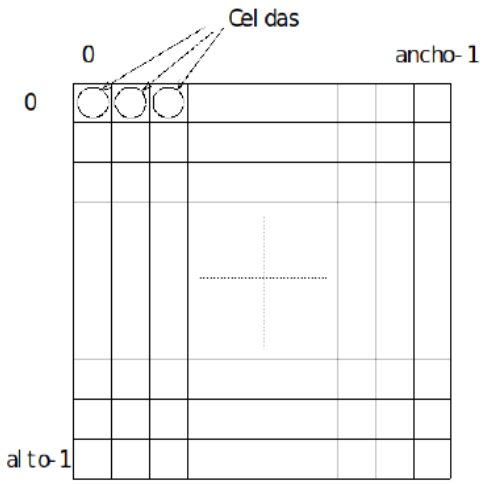


Figura 1.- Esquema de una imagen

Cada casilla de esta matriz representa un punto de la imagen y el valor guardado en ésta indica:

- En imágenes de **niveles de gris**: su nivel de luminosidad.
- En imágenes en color: su código de color (representación por tabla de color) o la representación del color en el esquema usado (RGB, IHV, etc.).

En esta práctica se trabajará con imágenes de niveles de gris, por lo que cada casilla contiene niveles de luminosidad que se representan con valores del conjunto {0,1, ..., 255} con la convención explicada anteriormente.

A modo de ejemplo, en la figura 2.A mostramos una imagen digital de niveles de gris que tiene 256 filas y 256 columnas. Cada uno de los 256 x 256 = 65.536 puntos contiene un valor entre 0 (visualizado en negro) y 255 (visualizado en blanco). En la figura 2.B mostramos una parte de la imagen anterior (10 filas y 10 columnas) en la que se pueden apreciar, con más detalle, los valores de luminosidad en esa subimagen.

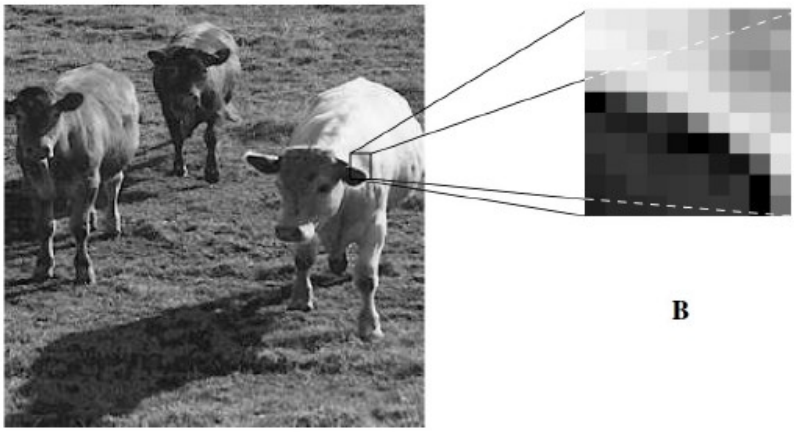


Figura 2.- A. Imagen real de 256 filas y 256 columnas. B. Una parte de esa figura, con 10 filas y 10 columnas

1.2 Procesamiento de imágenes digitales

El término **Procesamiento de Imágenes Digitales** incluye un vasto conjunto de técnicas cuyo objetivo final es la extracción y representación de información a partir de imágenes digitales. Constituye uno de los campos más atractivos de la Inteligencia Artificial, y en el ámbito científico se conoce con diferentes nombres: Computer Vision, Image Processing, etc., dependiendo del campo y ámbito de aplicación. Una definición informal, y bastante restringida, sería la siguiente: mediante *Procesamiento de Imágenes Digitales* nos referimos a un conjunto de operaciones que modifican la imagen original, dando como resultado una nueva imagen en la que se resaltan ciertos aspectos de la imagen original.

2. Formatos de imágenes

Las imágenes se almacenan en ficheros con un determinado formato. Existen numerosos formatos de imágenes y su descripción detallada puede encontrarse en la bibliografía específica de Procesamiento de Imágenes o en Internet. Así, para dar versatilidad a nuestra aplicación se requiere un conjunto adicional de funciones para que "rellenen" una imagen tomando la información de un fichero o para que guarden una imagen en un fichero. Entre los formatos de imágenes, vamos a trabajar con el formato PGM. Las funciones de lectura y escritura de imágenes van a constituir una biblioteca adicional, que se describe en esta sección.

El formato PGM constituye un ejemplo de los llamados formatos con cabecera. Estos formatos incorporan una cabecera en la que se especifican diversos parámetros acerca de la imagen, como el número de filas y columnas, número de niveles de gris, comentarios, formato de color, parámetros de compresión, etc.

2.1 Descripción del formato PGM

PGM es el acrónimo de Portable Gray Map File Format. Como hemos indicado anteriormente, el formato PGM es uno de los formatos que incorporan una cabecera. Un fichero PGM tiene, desde el punto de vista del programador, un formato mixto texto-binario: la cabecera se almacena en formato texto y la imagen en sí en formato binario. A continuación se muestra con más detalle la descripción del formato PGM.

1. Cabecera.

La cabecera está en formato texto y consta de:

- Un "número mágico" para identificar el tipo de fichero. Un fichero PGM que contiene una imagen digital de niveles de gris tiene asignado como identificador los dos caracteres P5.
- Un número indeterminado de comentarios.
- Número de columnas (c).
- Número de filas (f).
- Valor del mayor nivel de gris que puede tener la imagen (m).

Cada uno de estos datos está terminado por un carácter separador (normalmente un salto de línea).

2. Contenido de la imagen.

Una secuencia binaria de f x c bytes, con valores entre 0 y m. Cada uno de estos valores representa el nivel de gris de un píxel. El primero hace referencia al píxel de la esquina superior izquierda, el segundo al que está a su derecha, etc.

Algunas aclaraciones respecto a este formato:

- El número de filas, columnas y mayor nivel de gris se especifican en modo texto, esto es, cada dígito viene en forma de carácter.
- Los comentarios son de línea completa y están precedidos por el carácter #. La longitud máxima es de 70 caracteres. Los programas que manipulan imágenes PGM ignoran estas líneas.
- Aunque el mayor nivel de gris sea m, no tiene porqué haber ningún pixel con este valor. Este es un valor que usan los programas de visualización de imágenes PGM.

En la Figura 3 se muestra un ejemplo de cómo se almacena una imagen de 100 filas y 50 columnas en el formato PGM.

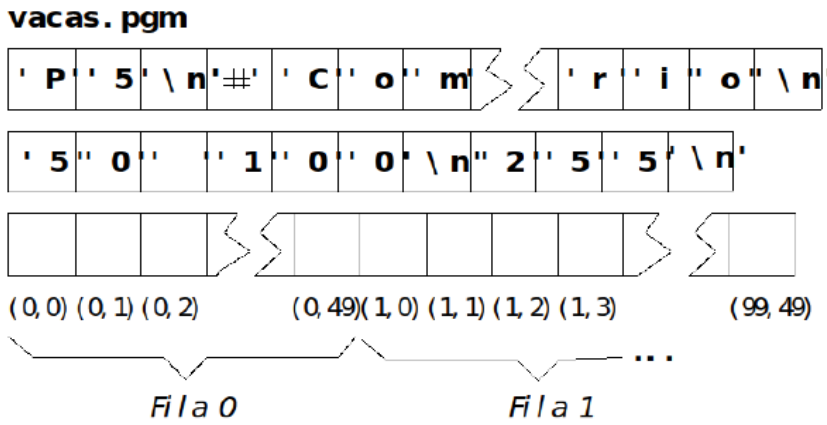


Figura 3. Almacenamiento de una imagen de niveles de gris con 100 filas y 50 columnas en un fichero con formato PGM (vacas.pgm).

2.2 Librería ImageIO

Tenemos a nuestra disposición ya implementada una librería para trabajar con imágenes PGM. Conocemos la especificación y no debemos preocuparnos por su implementación. Dispone entre otros de las siguientes funciones:

ReadPGMImage()

```
unsigned char * ReadPGMImage ( const char *  path,
                               int &         rows,
                               int &         cols
                             )
```

Lee una imagen de tipo PGM.

Parameters

path archivo a leer

rows Parámetro de salida con las filas de la imagen.

cols Parámetro de salida con las columnas de la imagen.

Returns

puntero a una nueva zona de memoria que contiene *filas* x *columnas* bytes que corresponden a los grises de todos los píxeles (desde la esquina superior izqda a la inferior drcha). En caso de que no no se pueda leer, se devuelve cero. (0).

Postcondition

En caso de éxito, el puntero apunta a una zona de memoria reservada en memoria dinámica. Será el usuario el responsable de liberarla.

WritePGMImage()

```
bool WritePGMImage ( const char *          path,
                     const unsigned char * datos,
                     const int             rows,
                     const int             cols
                   )
```

Escribe una imagen de tipo PGM.

Parameters

path archivo a escribir

datos punteros a los *f* x *c* bytes que corresponden a los valores de los píxeles de la imagen de grises.

rows filas de la imagen

cols columnas de la imagen

Returns

si ha tenido éxito en la escritura.

3.- El TDA imagen

A la hora de implementar programas de manipulación de imágenes digitales debemos plantearnos la conveniencia de utilizar un tipo de dato abstracto (TDA en lo sucesivo) para este propósito. Antes de nada, debemos considerar que previamente a su manipulación se requiere que la imagen resida en memoria y que, una vez que finalice el procesamiento, deberemos liberar la memoria que ocupaba. Hay dos operaciones básicas necesarias que son comunes a prácticamente la totalidad de procesos sobre una imagen: consultar y asignar el valor de un punto de la imagen, así como consultar el número de filas y columnas de la imagen. Todas las operaciones de procesamiento se pueden hacer en base a estas dos simples operaciones. Realmente no se requiere más para construir un programa de procesamiento de imágenes, aunque es útil disponer de dos operaciones complementarias: consultar el número de filas y consultar el número de columnas de una imagen.

Así, tras este breve análisis podemos plantear ya el TDA imagen. En primer lugar describiremos de forma abstracta el TDA (sección 3.1) a nivel de dominio y operaciones asociadas, para, posteriormente, proponer una representación de los objetos de tipo imagen y una implementación de las operaciones basadas en esa representación.

3.1 Descripción abstracta del TDA imagen

3.1.2 Dominio

Una imagen digital de niveles de gris, i, puede verse como una matriz bidimensional de **nf** filas y **nc** columnas, en la que en cada elemento, i (f, c), se guarda el nivel de luminosidad de ese punto. El tipo asociado a un objeto de estas características será el tipo **imagen**. Los valores de luminosidad están en el conjunto {0,1, ..., 255} de forma que el 0 indica mínima luminosidad (y se visualiza con el color negro) y el 255 la máxima luminosidad (y se visualiza con el color blanco). Los restantes valores indican niveles intermedios de luminosidad y se visualizan en distintos tonos de grises, siendo más oscuros cuanto menor sea su valor. El tipo asociado a cada una de las casillas de un objeto de tipo **imagen** será el tipo **pixel** (un pixel no es más que un unsigned char).

3.1.2 Especificación

Definimos una imagen digital i como:

```
Image i;
```

Las operaciones básicas asociadas son las siguientes:

1. Creación de una imagen.
2. Destrucción de una imagen.
3. Consultar el número de filas de una imagen.
4. Consultar el número de columnas de una imagen.
5. Asignar un valor a un punto de la imagen.
6. Consultar el valor de un punto de la imagen.

Algunas operaciones más complejas pueden ser:

1. Cargar una imagen desde un archivo.
2. Guardar una imagen en un archivo.

En el archivo Image.h puedes encontrar la especificación de estos y otros métodos, sus precondiciones y postcondiciones.

3.1.3 Ejemplos de uso del TDA imagen

Una vez definido de forma abstracta el TDA imagen, mostraremos algunos ejemplos de uso. Obsérvese que en ningún momento hemos hablado de su representación exacta en memoria ni de la implementación de las operaciones. De hecho, este es nuestro propósito, mostrar que puede trabajarse correctamente sobre objetos de tipo imagen independientemente de su implementación concreta.

Ejemplo: *invertir una imagen*

Se trata de invertir una imagen plana (obtener el negativo de una imagen). Queremos invertir una imagen existente alojada en la ruta origen y guardarla en la ruta destino:

```
Image image;
image.Load(origen);
for (int i=0; i < image.get_rows(); i++)
    for (int j=0; j < image.get_cols(); j++)
        image.set_pixel( i, j, MAX_BYTE_VALUE - image.get_pixel(i, j));
image.Save(destino);
```

Así, podemos invertir cualquier imagen, abstrayéndonos de su representación interna. De hecho, si entendemos que invertir una imagen es una operación habitual en el contexto del procesamiento de imágenes (parece que lo es), podemos encapsular su funcionamiento en un método de la propia clase:

```
Image image;
image.Load(origen);
image.Invert();
image.Save(destino);
```

De esta forma nos abstraemos de los detalles de la representación y la implementación. A través de la especificación de los métodos nos abstraemos del **cómo** y nos centramos en el **qué**.

3.2 Representación del tipo imagen

Aunque una imagen se vea o se contemple como una matriz bidimensional estática, no es ésta la representación más adecuada para los objetos de tipo imagen. La razón es que el compilador debe conocer las dimensiones de la matriz en tiempo de compilación y ésto no responde a nuestros propósitos al ser demasiado restrictivo. Así, debe escogerse una representación que permita reservar memoria para la imagen en tiempo de ejecución.

Resulta evidente que la naturaleza bidimensional de la imagen hace que una buena representación sea la de una matriz bidimensional dinámica. Esto nos permite acceder a cada punto de la imagen a través de la notación mediante índices y simplifica la programación de las operaciones.

Entre las distintas alternativas de implementación de matrices bidimensionales mediante vectores de punteros, la elección de una u otra dependerá de la especificación de las operaciones que vayamos a realizar sobre las imágenes y del conjunto de operaciones primitivas que proporcionemos al usuario del TDA.

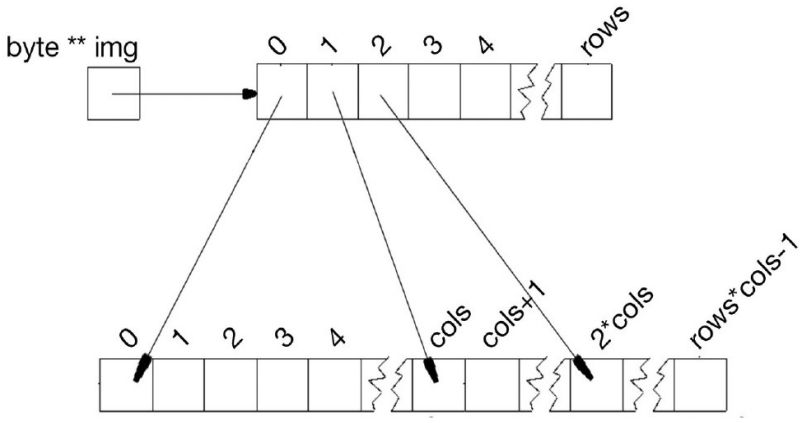


Figura 4. Representación sugerida

Ya que la librería ImageIO espera y devuelve un puntero a pixels, podemos plantearnos esta como una posible representación (ver Figura 4). Los pasos a seguir para obtener la matriz dinámica img serían:

1. Reservamos rows punteros a listas de bytes.
   `img = new pixel * [rows]`
2. Reservamos rows*cols bytes
   `pixel * total = new pixel [rows * cols]`
3. Asignamos a cada puntero a fila la posición del primer byte correspondiente.
   ```
   for (int i=0; i < rows; i++)
      img[i] = total + (i*cols);
   ```

En la la parte privada de la clase Image tenemos, además de la matriz bidimensional, el tamaño de la matriz:

```
int rows;  // Numero de filas de la imagen
int cols;  // Numero de columnas de la imagen
pixel **img; // La imagen en si: una matriz dinamica 2D de pixels
```

Podemos agrupar los tipos presentados y los prototipos de las funciones en un fichero de cabecera, al que llamaremos image.h. La idea es tener una cabecera con las especificaciones por un lado y la implementación abstraída y contenida en la librería libimage.a por otro, generada a partir de los archivos uente image.cpp (constructores y operaciones primitivas) e imageop.cpp (operaciones más complejas). Así, el usuario del TDA dispondrá de la biblioteca (con las funciones compiladas) y del fichero de cabecera asociado (con los tipos descritos anteriormente y los prototipos de las funciones públicas de la biblioteca).

Nota: Una representación alternativa a la actual es la siguiente: `pixel ** img` ahora apunta a una lista de `pixel *` no necesariamente consecutivos en memoria. Podemos, o bien llamar a `row[i] = new pixel[cols]` por cada fila (ver Figura 5) o bien, más sencillo, mantener la implementación como hasta ahora (ver apartado 3.2) pero dejar de asumir que `img[i+1] == img[i]+cols` tan pronto como finalicemos la inicialización de la imagen. Si lo hemos hecho todo correctamente, el único implicado en este cambio debería ser `bool Image::Save (const char * file_path) const`.

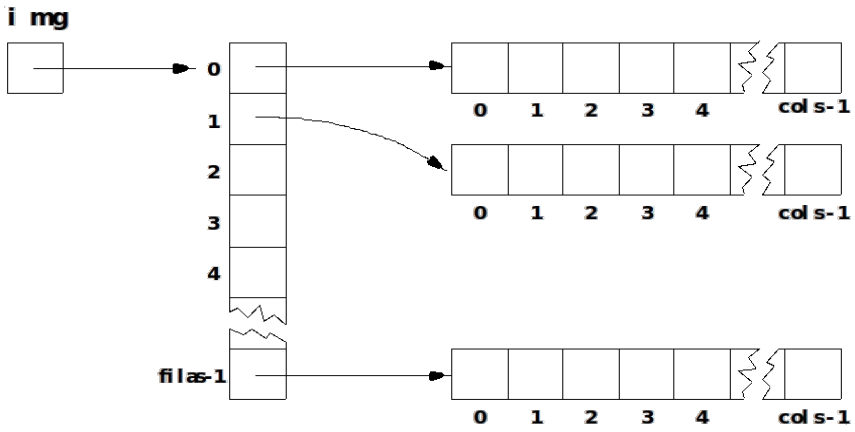


Figura 5. Representación alternativa del contenido de una imagen

4.- TDA Video

En esta sección se dan los detalles acerca del TDA Video que el estudiante deberá especificar e implementar en esta práctica.

Un vídeo puede verse como una secuencia temporal de im.genes (o fotogramas). Los ejemplos de vídeos que vamos a llevar a cabo son sobre secuencias de imágenes de niveles de gris.

4.1.- Dominio

Un vídeo digital, V, puede contemplarse como un vector con n fotogramas (imágenes digitales que han sucedido a lo largo de un espacio temporal de una escena). En un vídeo cada elemento, `V(t) (0<=t<n)`, almacena una imagen digital asociada al instante *t*.

4.2.- Operaciones

Dado una vídeo digital `v`, definido por ejemplo como:

```
Video v;
```

las operaciones básicas asociadas son las siguientes:

1. Creación de un vídeo.
2. Destrucción de un vídeo.
3. Consultar el número de fotogramas de un vídeo.
4. Asignar/consultar una imagen dentro del vídeo
5. Insertar un nuevo fotograma.
6. Borrar un fotograma.
7. Leer un vídeo en disco.
8. Almacenar un vídeo en disco.

A continuación se describen con detalle.

Creación de un vídeo. Podemos construir un vídeo vacío, como copia de otro, o crearlo con una secuencia de n imágenes vacías.

```
Video::Video (); // Constructor por defecto
```

```
Video::Video (const Video & J); // Constructor de copias

Video::Video(int n);
```

Operador de tipo constructor.

Propósito: Crear un vídeo en memoria con posibilidad para albergar n imágenes. Reserva memoria para alojar el vídeo.

Precondiciones: n>0

Postcondiciones:

1. El vídeo creado tiene n casillas.
2. El vídeo creado contiene valores "basura" (no está inicializado).

Ejemplo de uso: `Video V(100); Video V; Video V(J);`

Destrucción de un vídeo.

```
Video::~Video();
```

Operador de tipo destructor.

Propósito: Liberar los recursos ocupados por el vídeo.

Postcondiciones:

1. Devuelve: nada.
2. El vídeo destruido no puede volver a usarse, salvo que se vuelva a realizar sobre él otra operación de construcción.

Consultar el número de fotogramas del vídeo

```
int Video::size () const;
```

Propósito: Calcular el número de fotogramas del vídeo.

Postcondiciones:

1. Devuelve: Numero de fotogramas del vídeo.
2. El vídeo no se modifica.

Ejemplo de uso: `V.size();`

Consultar/Asignar una imagen al vídeo

```
Imagen &  Video::operator[](int foto);

const Imagen &  Video::operator[](int foto)const;
```

Propósito: A través del operador [] el usuario puede consultar un fotograma del vídeo.

Precondiciones:

0 <= n < size ()

Postcondiciones:

Devuelve: la imagen asociada al fotograma foto-ésimo del vídeo.

Ejemplo de uso: `V[10]=I; //siendo I una imagen y V un vídeo no constante`

Insertar un nuevo fotograma

```
void Video::Insertar(int k, const Imagen&I);
```

Propósito: Inserta un nuevo fotograma I en la posición k.

Precondiciones: k debe indicar una posición válida dentro del vídeo, es decir, en el rango [0,V.size()].

Postcondiciones:

El vídeo es modificado teniendo un nuevo fotograma más.

Ejemplo de uso:

```
Imagen I;
Video V;
….
V.Insertar(0,I);
```

Borrar un fotograma

```
void Video::Borrar(int k);
```

Propósito: Eliminar el fotograma de la posición dada por un entero k.

Precondiciones: k debe ser una posición válida dentro del vídeo, es decir, que pertenezca al rango [0,V.size()-1].

Postcondiciones:

El vídeo es modificado teniendo un fotograma menos.

Ejemplo de uso:

```
Video V;
…
V.Borrar(0);
```

Leer un vídeo

```
bool Video::LeerVideo(const string & path);
```

Propósito: Leer un vídeo de disco.

Precondiciones: en el directorio path debe almacenarse una secuencia temporal de fotogramas.

Postcondiciones:

Devuelve true si la lectura tiene éxito y false en caso contrario.

Ejemplo de uso:

```
Video V;
…
V.LeerVideo("datos/videos/missa/");
```

Escribir un vídeo

```
bool Video::EscribirVideo(const string & path, const string &prefijo)const;
```

Propósito: Escribe un vídeo de disco.

Precondiciones: en el directorio path se almacenará la secuencia temporal de fotogramas. Los fotogramas tomarán como nombre <prefijo>_01.pgm .. <prefijo>_0n.pgm siendo n el número de fotogramas.

Postcondiciones:

Devuelve true si la escritura tiene éxito y false en caso contrario.

Ejemplo de uso:

```
Video V;
…
V.EscribirVideo("datos/videos/missa/");
```

4.3.- Representación del TDA Vídeo

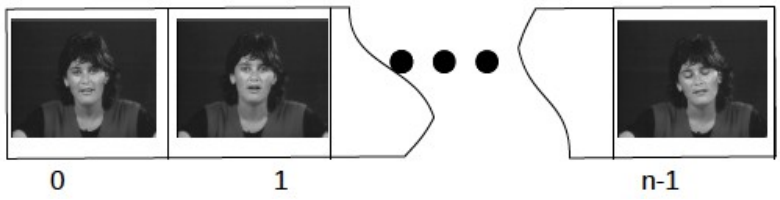
Existen diferentes representaciones que podemos hacer del TDA Vídeo:

- Vector Dinámico.
- Lista con celdas enlazadas.
- Listas con celdas doblemente enlazadas y cabecera.
- Etc.

La elección que hagamos debe ser analizada sobre qué coste en eficiencia tienen las operaciones que caracterizan al TDA Vídeo. Así, las operaciones que van a definir la eficiencia son las operaciones Insertar y Borrar. En nuestro caso, vamos a optar por un *Vector Dinámico*. Para ello, usaremos la *clase vector de la STL*.

4.3.1.- Vector Dinámico

Un vector dinámico no es más que un vector con la posibilidad de crecer y decrecer en tiempo de ejecución según las necesidades del programa. En nuestro caso, un vídeo vendría representado por un vector de la STL instanciando a Image (TDA presentado en la sección 3).



Interface: video.h

```
1.  #ifndef VIDEO__H
2.  #define VIDEO__H
3.  #include <vector>
4.  #include "image.h"
5.  using namespace std;
6.  class Video{
7.    private:
8.      vector<Image> seq;
9.    //operaciones
```

```
10.    .....
11.  };
12.  #endif
```

4.4.- Forma de almacenar un vídeo

Ya que un vídeo lo podemos interpretar como una secuencia de imágenes, almacenar un vídeo en disco significa almacenar los fotogramas que lo forman. Así, cada fotograma lo almacenaremos en formato PGM. Para saber qué imágenes forman un vídeo, los fotogramas se almacenan en disco con nombres de ficheros que siguen la siguiente nomenclatura:

```
prefijo<n>.pgm
```

donde n va desde 001 a 00n.

Si queremos ver un video podemos usar el paquete ImageMagick. En particular ejecutando:

```
display prefijo*.pgm
```

O alternativamente usando el comando:

```
animate prefijo*.pgm
```

5.- Práctica a realizar

Para poder llevar a cabo las siguientes tareas deberías previamente haber creado el TDA Vídeo a través de la clase Vídeo, cuya implementación se proporciona a través de docencia virtual. Una vez construida la clase, se han de hacer los dos programa de prueba que se indican a continuación.

Rebobinar un video.

Vamos a generar un programa que, a partir de un vídeo, obtenga el vídeo invertido. Para ello, se debe implementar la función Rebobinar dentro del programa:

```
Video Rebobinar (const Video &v);
```

Esta función tiene como parámetro de entrada un vídeo constante y devolverá un nuevo vídeo que será el vídeo invertido. Esto quiere decir que el fotograma v.size()-1 se podrá en la posición 0 del video rebobinado, el v.size()-2 en la posición 1, y así hasta llegar al fotograma 0 que se colocará en la posición v.size()-1.

Deberéis generar un programa cuyos parámetros de entrada son:

- directorio donde se encuentran los fotogramas del vídeo de entrada, y
- directorio donde se escribe el vídeo rebobinado.

Un ejemplo de llamada podría ser el siguiente:

```
./bin/rebobinar datos/videos/missa/ datos/videos/reb_missa/
```

Morphing.

El morphing se usa para cambiar una imagen en otra o para proporcionar una transición suave de una imagen a otra creando la ilusión de una transformación. El término está tomado de la industria de los efectos especiales. Una forma de hacer un morphing simple es establecer un proceso para

incrementar o decrementar los valores de los píxeles a partir de una imagen fuente hasta que igualen los valores de los correspondientes pixeles en una imagen destino.

En esta práctica vamos a realizar un morphing muy básico. Para ello, sea nf el número de fotogramas que queremos obtener e I1 e I2 las imágenes que queremos transformar.

El vídeo v en la posición i-.sima contendrá la imagen que sigue la siguiente fórmula:

```
I1* (1-i/nf) + (I2*i)/nf
```

Una forma más compleja para obtener un morphing más riguroso se puede ver en el Apéndice A.

El estudiante creará el programa morphing, que tendrá la siguiente función:

```
Video morphing (const Image &I1, const &Image &I2, int nf);
```

Los parámetros que recibe el programa son los siguientes:

- fich_orig es el nombre del fichero que contiene la imagen inicial de la que se parte.
- fich_rdo es el nombre del fichero que contiene la imagen final a la que se pretende llegar.
- Directorio donde almacenar el vídeo creado.
- Número de pasos para realizar la transformación.

Un ejemplo de llamada sería:

```
./bin/morphing datos/imagenes/Pedro.pgm datos/imagenes/Azucena.pgm
                        datos/videos/morph 40
```

4.- Documentación y entrega

Toda la documentación de la práctica se incluirá en el propio Doxygen generado. Para ello se utilizarán tanto las directivas Doxygen de los archivos .h y .cpp como los archivos .dox incluidos en la carpeta doc. Leed detenidamente la documentación de ejemplo para ver cómo desarrollarla correctamente.

1. La documentación debe incluir:

- TODOS los métodos y clases debidamente documentados con la especificación completa. Se valorarán positivamente descripciones exhaustivas y el uso de elementos adicionales como fórmulas e imágenes. La documentación disponible de ejemplo puede y DEBE ser completada.
- Cada tarea de la práctica tiene un ejecutable asociado que debe ser descrito en la página principal o en una página *ad hoc*. Se valorará positivamente el uso de vídeos para ilustrar el funcionamiento de los programas.
- Todas las páginas de la documentación generada deben ser completas y estar debidamente descritas.

2. A considerar:

- Deben respetarse todos los conocimientos adquiridos en los temas de abstracción y eficiencia, en especial el principio de ocultamiento de información y las distintas estrategias de abstracción.
- Una solución algorítmicamente correcta pero que contradiga el punto anterior, será considerada como errónea.
- El número de métodos que interactúen directamente con la representación interna del TDA deberá mantenerse al mínimo posible y deberá estar debidamente justificado su uso.

3. Código:

- Se dispone de la siguiente estructura de ficheros:

```
/
        estudiante/                 (Aquí irá todo lo que desarrolle el estudiante)
                doc/                (Imágenes y documentos extra para Doxygen)
                include/            (Archivos cabecera)
                src/                (Archivos fuente)
                datos/              (Datos con imágenes y vídeos)
                CMakeList.txt       (Instrucciones Cmake si se quiere usar)
                Doxyfile.in         (Archivo de configuración de Doxygen)
```

- Es muy importante (si se va a usar) leer detenidamente y entender qué hace CMakeList.txt.
- El archivo Doxyfile.in no tendríais porqué tocarlo para un uso básico, pero está a vuestra disposición por si queréis añadir alguna variable (no cambiéis las existentes).
- Crear un archivo llamado submission.zip que incluya TODO lo necesario para subirlo a Prado para la entrega.

4. Elaboración y puntuación:

- La práctica se realizará **por parejas**. Los nombres completos se incluirán en la descripción de la entrega en Prado. Ambos integrantes de la pareja deben subir el archivo a Prado.
- Desglose:
    - (0.4) Ejercicios 1-2.
    - (0.1) Documentación
- La fecha límite de entrega aparecerá en la subida a Prado.

APÉNDICE A

El morphing es un efecto de transición suave y progresiva entre dos (o más) imágenes. Está basado en algunas ideas muy simples:

- Una transformación geométrica, definida por un conjunto de puntos de origen, S, y otro de destino, D.
- La transformación geométrica está graduada, entre las posiciones de origen y de destino: 0% posiciones de origen, 100% posiciones de destino y 50% término medio entre ambos.
- El color de un píxel es una media ponderada, entre la imagen origen y destino, según el

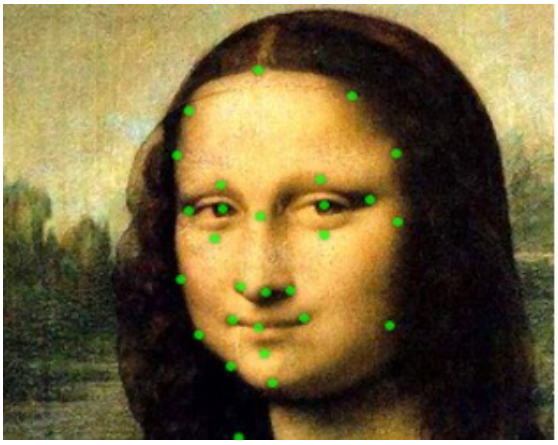
grado de la transformación.

Analíticamente:

S= (s1, s2, ..., sn), D= (d1, d2, ..., dn)

En el paso g, los puntos intermedios son: $i_k = (1-g) \cdot s_k + g \cdot d_k$

Transformar A, moviendo los puntos de S a I -> R1.
Transformar B, moviendo los puntos de D a I -> R2.



Puntos de origen S

Puntos de destino D